chapter-08-03

March 14, 2025

Run in Google Colab

1 Deep Learning in Unsupervised Learning

1.1 Generative Adversarial Networks (GANs)

1.1.1 Introduction

- Generative Adversarial Networks (GANs) represent one of the most important development in the field of artificial intelligence, particularly in the domain of generative modeling.
- Introduced by Ian Goodfellow and his colleagues in 2014, GANs have revolutionized the way machines can generate realistic images, texts, and other forms of data.
- The essence of GANs lies in their unique architecture and training methodology, which involves a dueling process between two neural networks: the Generator and the Discriminator.

1.1.2 Architecture of GANs

The architecture of a GAN is composed of two main components: the Generator and the Discriminator, each playing a distinct role in the generative process.

1. **Generator**: The Generator network takes random noise as input and transforms it into data that resemble the target distribution. The goal of the Generator is to produce outputs indistinguishable from real data, essentially "fooling" the Discriminator. It learns to generate more realistic data over time as it adjusts its parameters based on the feedback from the Discriminator.
2. **Discriminator**: The Discriminator network acts as a classifier, tasked with distinguishing between real data (from the training set) and fake data produced by the Generator. It is trained to improve its accuracy in detecting the Generator's outputs as fake.

These two networks engage in a continuous game, with the Generator striving to produce increasingly convincing data, while the Discriminator becomes better at distinguishing real from fake. This adversarial process drives the improvement of both networks, leading to the generation of highly realistic data.

1.1.3 Training Phase

Training a GAN involves alternating between training the Discriminator and the Generator, with both networks being updated based on their performance in each iteration.

1. **Training the Discriminator**: In this step, the Discriminator is trained with a batch of real data labeled as real and a batch of fake data generated by the Generator, labeled as fake. The goal is to maximize its accuracy in correctly classifying real and fake data. This is typically done using a binary cross-entropy loss function.
2. **Training the Generator**: After updating the Discriminator, the Generator is trained. The key difference here is that the Generator's output is passed to the Discriminator, and the Generator is updated based on how well the Discriminator was able to distinguish its output from real data. The Generator's objective is to minimize the chance of the Discriminator correctly identifying its outputs as fake.

1.1.4 Backpropagation of Error During Training

The backpropagation algorithm plays a crucial role in updating the weights of both the Generator and Discriminator networks. During the training phase, backpropagation is used to calculate the gradients of the loss function with respect to the network parameters, allowing for the optimization of these parameters.

1. **For the Discriminator**, backpropagation calculates the gradient of the loss function that measures its *ability to distinguish between real and fake data.* The weights are then updated to improve its classification accuracy.
2. **For the Generator**, the process is slightly more complex because *its success is measured by the Discriminator's failure to recognize its outputs as fake.* Thus, the gradient of the Discriminator's loss function with respect to the Generator's weights is computed, essentially "tricking" the Discriminator. The Generator's weights are updated to produce outputs that are more likely to be classified as real by the Discriminator.

This iterative process of backpropagation and updating continues until a stopping criterion is met, which could be a certain number of iterations or when the Discriminator's accuracy reaches a specific threshold, indicating that the Generator produces sufficiently realistic data.

1.1.5 Generative Vs Discriminative Models

- If you have delved into neural networks, it's probable that the majority of the examples you encountered utilized ***discriminative*** models. Conversely, generative adversarial networks belong to a distinct category known as ***generative*** models.
- Discriminative models are typically employed for most tasks involving supervised learning, whether for classification or regression. Consider a scenario where the objective is to develop a model capable of identifying handwritten digits ranging from 0 to 9. In this case, one would employ a dataset comprising images of handwritten digits, each tagged with a label that denotes the represented digit.
- Throughout the model training phase, an algorithm would be applied to tweak the model's parameters. The aim here is to reduce a loss function to a minimum, enabling the model to grasp the ***probability distribution of the output based on the given input***. Once the training is completed, the model can then be used to identify new images of handwritten digits by predicting the digit that most likely matches the input, as depicted in the following illustration:

image source: RealPython see References and Credits Section

- You can think of discriminative models for classification tasks as structures that leverage training data to identify the dividing lines between different classes. These models then apply these learned boundaries to distinguish an input and assign it to a specific category.
- In terms of mathematics, discriminative models are adept at **learning the conditional probability** $P(y|x)$, which is the probability of the output y given the input x.
- Beyond neural networks, discriminative models can also take the form of logistic regression models and support vector machines (SVMs).
- On the other side, **generative models** such as GANs are developed to model the way a dataset is produced through a probabilistic framework. By drawing samples from a generative model, it becomes possible to **create new pieces of data**. While discriminative models find their application in supervised learning, generative models are typically employed with datasets that do not have labels, representing a type of unsupervised learning.
- For instance, using a dataset of handwritten digits, a generative model could be trained to produce new digit images. During its training, an algorithm would be utilized to refine the model's parameters with the aim of reducing a loss function and capturing the **probability distribution of the dataset**. Following the training, this model would be capable of generating new digit samples.

image source: RealPython see References and Credits Section

- Generative models typically incorporate a stochastic, or random, factor to produce new samples.
- This randomness is essential for the variety in the samples created by the model. The random inputs fueling the generator come from what is known as a latent space, where the vectors serve as a condensed version of the samples that are generated.
- In contrast to discriminative models, generative models are focused on learning the **probability** $P(x)$ of the input data x. With this understanding of the input data's distribution, they have the capability to generate new instances of data.

1.1.6 A Simple Toy Model

- As we have already said, generative adversarial networks consist of an overall structure composed of two neural networks, one called the generator and the other called the discriminator.
- The role of the generator is to estimate the probability distribution of the real samples in order to provide generated samples resembling real data. The discriminator, in turn, is trained to estimate the probability that a given sample came from the real data rather than being provided by the generator.
- These structures are called generative adversarial networks because the generator and discriminator are trained to compete with each other: the generator tries to get better at fooling the discriminator, while the discriminator tries to get better at identifying generated samples.
- To understand how GAN training works, consider a toy example with a dataset composed of two-dimensional samples (x_1, x_2), with x_1 in the interval from 0 to 2π and $x_2 = \sin(x_1)$.

To train D, at each iteration you label some real samples taken from the training data as 1 and some generated samples provided by G as 0. This way, you can use a conventional supervised training framework to update the parameters of D in order to minimize a loss function, as shown in the following scheme:

For each batch of training data containing labeled real and generated samples, you update the parameters of D to minimize a loss function. After the parameters of D are updated, you train G to produce better generated samples. The output of G is connected to D, whose parameters are kept frozen, as depicted here:

Now that you know how GANs work, you're ready to implement your own using PyTorch…

First of all as usual we have to import python libraries. The `torch` library and its neural network module `nn`, along with the `math` and `matplotlib.pyplot` libraries. `torch` is a popular deep learning library offering rich functionalities for tensor operations, automatic differentiation, and neural network design. The `nn` module within `torch` provides essential building blocks for creating neural networks, such as layers and activation functions. The `math` library is used for mathematical functions, and `matplotlib.pyplot` is a plotting library for visualizing data, which is useful for tasks like displaying the results of model training or data analysis.

[1]:
```
import torch
from torch import nn

import math
import matplotlib.pyplot as plt
```

The code `torch.manual_seed(123)` sets the seed for generating random numbers in PyTorch to `123`. This ensures that the random numbers generated by PyTorch functions are deterministic and reproducible across runs of the program. Setting a manual seed is crucial for experiments where reproducibility is necessary, as it ensures that the random elements of the computations (such as weight initialization in neural networks) are consistent each time the code is executed.

[2]:
```
torch.manual_seed(123)
```

[2]:
```
<torch._C.Generator at 0x2a04ad09410>
```

Preparing the Training Data

Now we are going to create a dataset for a PyTorch training session, where the objective is to learn the sine function from its input. The following code initializes a dataset with 1024 samples (`train_data_length`), where each sample has two dimensions. The first dimension is randomly generated angles (in radians), and the second dimension is the sine of these angles. In other words, the training data is composed of pairs (x_1, x_2) so that x_2 consists of the value of the sine of x_1 for x_1 in the interval from 0 to 2π. Labels for this dataset are initialized to zeros (`train_labels`), although they're not used further in this snippet. Finally, it creates a list of tuples (`train_set`), where each tuple contains a data point and its corresponding label, ready for use in training a model.

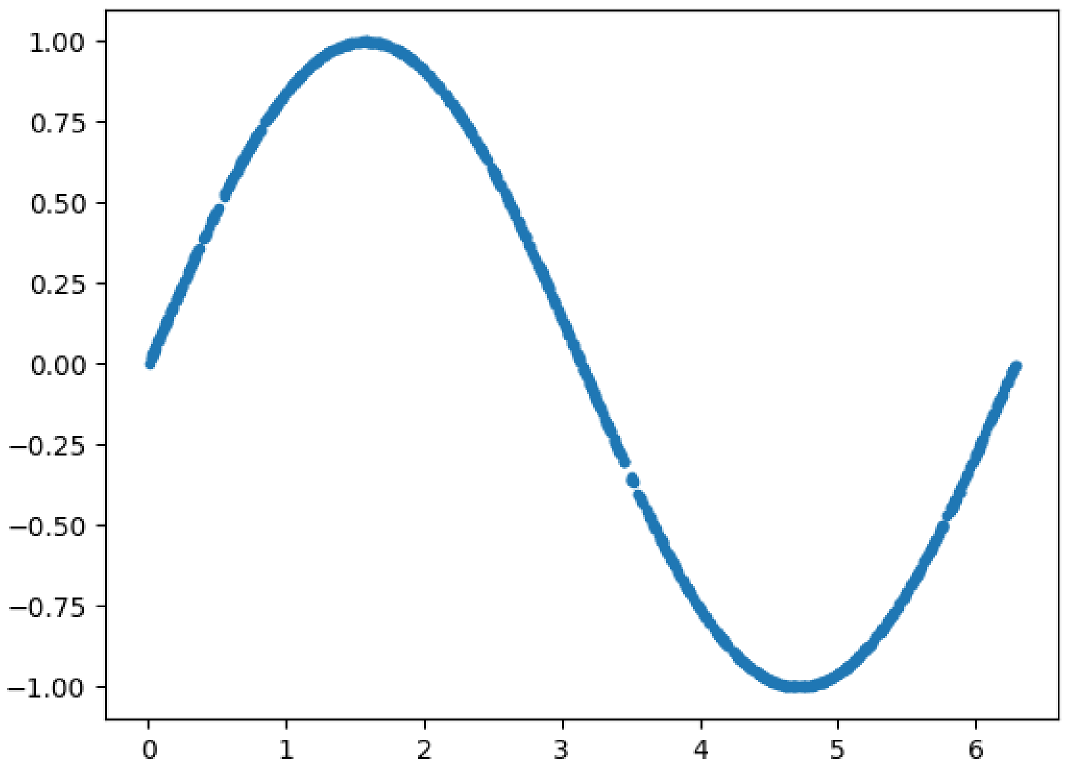
[3]:
```
# Set the total number of data points in the training set
train_data_length = 1024

# Initialize a tensor of zeros with shape (1024, 2) to hold the training data
train_data = torch.zeros((train_data_length, 2))
```

```
# Populate the first column of train_data with random angles between 0 and 2
train_data[:, 0] = 2 * math.pi * torch.rand(train_data_length)

# Calculate the sine of the angles (first column) and store the results in the␣
↪second column
train_data[:, 1] = torch.sin(train_data[:, 0])

# Initialize a tensor of zeros to serve as labels for the training data. Note␣
↪that
# the tensor of labels, is required by PyTorch's data loader. Since GANs make␣
↪use
# of unsupervised learning techniques, the labels can be anything. They won't␣
↪be used, after all.
train_labels = torch.zeros(train_data_length)

# Combine the training data and labels into a list of tuples, creating the␣
↪final training dataset
train_set = [
    (train_data[i], train_labels[i]) for i in range(train_data_length)
]
```

You can examine the training data by plotting each point

[4]:
```
plt.plot(train_data[:, 0], train_data[:, 1], ".")
```

[4]:
```
[<matplotlib.lines.Line2D at 0x2a04aefa8c0>]
```



Now we initialize a DataLoader for the training data with a specified batch size of 32. The DataLoader is a PyTorch utility that provides an iterable over the given dataset (`train_set` in this case), allowing for easy access to batches of data. The `batch_size` parameter defines how many samples per batch to load, and `shuffle=True` ensures that the data is shuffled at every epoch, which helps in reducing model overfitting by providing a more generalized learning.

[5]:
```
batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

Note. *A PyTorch DataLoader is a powerful tool designed to simplify data manipulation during the training of neural networks. It allows you to efficiently manage datasets by automating data batching, shuffling, and distribution across multiple processes. DataLoaders are particularly useful when dealing with large datasets that cannot fit into memory, as they enable seamless and on-the-fly data loading and transformation, thereby optimizing the training process and improving model performance.*

For a detailed understanding of PyTorch DataLoaders, the official PyTorch documentation is the best reference. It provides comprehensive insights into DataLoader functionalities, including examples and guidelines on how to customize data loading for different scenarios. This resource is invaluable for developers looking to leverage DataLoaders for efficient data handling in neural network training. Visit the PyTorch DataLoader *documentation for more information.*

Implementing the Discriminator

In PyTorch, to build neural network models, you create classes derived from `nn.Module`. This approach requires you to define a class for the discriminator, following principles of Object-Oriented Programming (OOP) in Python 3. The discriminator, designed to differentiate between real and generated data, accepts 2D inputs to output a probability indicating if the input is from the actual dataset. The following example illustrates constructing such a discriminator model.

[6]:
```
# Define the Discriminator class which is a subclass of nn.Module
class Discriminator(nn.Module):
    def __init__(self):
        # Initialize the superclass nn.Module
        super().__init__()

        # Construct the neural network model
        self.model = nn.Sequential(
            # First linear layer taking a 2D input and outputting 256 features
            nn.Linear(2, 256),
            # ReLU activation function to introduce non-linearity
            nn.ReLU(),
            # Dropout layer to prevent overfitting by randomly setting input␣
 ↪units to 0 with a probability of 0.3
            nn.Dropout(0.3),
            # Second linear layer reducing the feature size from 256 to 128
            nn.Linear(256, 128),
            # ReLU activation function
            nn.LeakyReLU(),
            # Another dropout layer for regularization
            nn.Dropout(0.3),
            # Third linear layer reducing the feature size from 128 to 64
            nn.Linear(128, 64),
            # ReLU activation function
            nn.LeakyReLU(),
            # Final dropout layer
            nn.Dropout(0.3),
            # Final linear layer reducing the feature size from 64 to 1
            nn.Linear(64, 1),
            # Sigmoid activation function to output a probability between 0 and␣
 ↪1
            nn.Sigmoid(),
        )

    # Define the forward pass through the network
    def forward(self, x):
        # Pass the input x through the sequential model and return the output
        output = self.model(x)
        return output
```

For more detailed explanations and tutorials on how to use PyTorch's `nn.Module` to create models, you can refer to the official PyTorch documentation. This resource provides comprehensive guidance on building neural networks with PyTorch, including various modules and functions available within the framework.

[7]:
```
discriminator = Discriminator()
```

Implementing the Generator

As we have explained, in Generative Adversarial Networks (GANs), the generator is a neural model designed to produce data similar to the training set, using inputs sampled from a latent space. In our example, this involves a model that takes two-dimensional random vectors as input and outputs data points aiming to mimic the real dataset. Like the discriminator's setup, creating a generator involves defining a Generator class that extends `nn.Module`, encapsulating the network's architecture. Subsequently, an instance of this Generator class is created for use.

[8]:
```
# Define the Generator class which inherits from nn.Module
class Generator(nn.Module):
    def __init__(self):
        super().__init__()  # Initialize the superclass (nn.Module)
        # Define the model architecture using a Sequential container
        self.model = nn.Sequential(
            nn.Linear(2, 16),  # First linear transformation from 2D input to␣
 ↪16D
            nn.LeakyReLU(),         # ReLU activation function for non-linearity
            nn.Linear(16, 32), # Second linear transformation from 16D to 32D
            nn.LeakyReLU(),         # Another ReLU activation function
            nn.Linear(32, 2),  # Final linear transformation from 32D back to 2D
        )

    # Define the forward pass method
    def forward(self, x):
        output = self.model(x)  # Pass the input x through the sequential model
        return output           # Return the model output

# Instantiate the Generator class
generator = Generator()
```

Training the Models

Before training the models, you need to set up some parameters to use during training:

[9]:
```
# sets the learning rate (lr), which you'll use to adapt the network weights
lr             = 0.001
# sets the number of epochs (num_epochs), which defines how many repetitions
# of training using the whole training set will be performed.
num_epochs     = 1000
```

```
#  assigns the variable loss_function to the binary cross-entropy function␣
↪BCELoss(),
# which is the loss function that you'll use to train the models.
loss_function = nn.BCELoss()
```

The binary cross-entropy function is a suitable loss function for training the discriminator because it considers a binary classification task. It's also suitable for training the generator since it feeds its output to the discriminator, which provides a binary observable output.

> **REMIND** *Binary Cross Entropy (BCE) is a loss function used in binary classification tasks in neural networks. It measures the difference between the true labels and the predicted probabilities of the positive class. BCE encourages the model to output probabilities close to 0 or 1, penalizing predictions that are confident but wrong. It's particularly useful for models ending with a sigmoid activation function, optimizing the parameters to minimize the discrepancy between actual and predicted labels, thus guiding the training towards more accurate predictions.*

These lines initialize optimizers for both the discriminator and generator models in a GAN setup using the Adam optimization algorithm. Each optimizer is configured to update the respective model's parameters with a learning rate defined by `lr`. The `discriminator.parameters()` and `generator.parameters()` methods collect all trainable parameters of each model, allowing the optimizer to adjust them during training to minimize the models' loss functions.

> **REMIND** *The Adam optimization algorithm is a method used to update the weights in neural networks more efficiently. It combines ideas from two other optimizations - RMSprop and Momentum - to calculate adaptive learning rates for each parameter. Adam is particularly useful for large datasets and high-dimensional spaces because of its efficient computation of gradient descent, making it faster and more precise in reaching the minimum loss. This makes Adam a popular choice for training deep learning models, especially when dealing with complex data and neural network architectures. For a deeper understanding of the Adam optimization algorithm and its practical applications, you might find the original paper by Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", to be a valuable resource. Here you can find a deep dive into the math of this method. Additionally, the TensorFlow and PyTorch documentation provide excellent overviews and implementation details that are useful for applying Adam in deep learning projects.*

[10]:
```
optimizer_discriminator  = torch.optim.Adam(discriminator.parameters(), lr=lr)
optimizer_generator      = torch.optim.Adam(generator.parameters(), lr=lr)
```

Finally, you need to implement a training loop for the Generative Adversarial Network, iterating through epochs to train both the discriminator and generator models. It processes batches of real and generated samples, updating model weights to minimize respective loss functions. The discriminator learns to differentiate real data from fake, while the generator aims to produce increasingly realistic samples. Losses are calculated using a specified loss function and backpropagated to update the models. Progress is logged at intervals, showing the discriminator's and generator's loss at the end of certain epochs.

> **NOTE** Recall that, with PyTorch, when initializing `optimizer` you explicitly tell it what parameters (tensors) of the model it should be updating. The gradients are

"stored" by the tensors themselves (they have a `grad` and a `requires_grad` attributes) once you call `backward()` on the loss. After computing the gradients for all tensors in the model, calling `optimizer.step()` makes the optimizer iterate over all parameters (tensors) it is supposed to update and use their internally stored grad to update their values. Optimizer just iterates through the list of parameters (tensors) it received on initialization and everywhere a tensor has `requires_grad=True`, it subtracts the value of its gradient stored in its `.grad` property (simply multiplied by the learning rate in case of SGD). It doesn't need to know with respect to what loss the gradients were computed it just wants to access that `.grad` property so it can do `x = x - lr * x.grad`.

see also this tutorial and this stack overflow question

NOTE The `zero_grad()` function in PyTorch is used to reset the gradients of all model parameters to zero. It's typically called before performing backpropagation (`.backward()`) during training. This is necessary because gradients accumulate by default for each `.backward()` call to support training of recurrent neural networks. Resetting gradients ensures that the model updates are based only on the most recent loss computation and not affected by previous iterations.

[23]:
```
latent_space_samples_for_test = torch.randn(100, 2)
# As is generally done for all neural networks, the training process consists␣
 ↪of two loops,
# one for the training epochs and the other for the batches for each epoch.
# Loop through each epoch.
for epoch in range(num_epochs):
    # Enumerate over the training loader to get batches of real samples. You␣
 ↪get the real samples of the
    # current batch from the data loader and assign them to real_samples.
    for n, (real_samples, _) in enumerate(train_loader):
        # This instruction creates a tensor filled with ones, having a shape␣
 ↪defined by (batch_size, 1).
        # This tensor is used to label the real samples which are typically␣
 ↪labeled with 1s, while generated (fake)
        # samples are labeled with 0s. Here, batch_size determines the number␣
 ↪of samples per batch.
        # Notice that the first dimension of the tensor has the number of␣
 ↪elements equal to batch_size.
        # This is the standard way of organizing data in PyTorch, with each␣
 ↪line of the tensor representing
        # one sample from the batch.
        real_samples_labels = torch.ones((batch_size, 1))
        # Generate random points in the latent space. You create the generated␣
 ↪samples by storing random data
        # in latent_space_samples, which you then feed to the generator to␣
 ↪obtain generated_samples.
        latent_space_samples = torch.randn((batch_size, 2))
        # Generate fake samples from the latent space samples.
        generated_samples = generator(latent_space_samples)
        # Assign labels of 0 to generated (fake) samples.
        generated_samples_labels = torch.zeros((batch_size, 1))
        # Concatenate real and generated samples to form a mixed dataset.
        all_samples = torch.cat((real_samples, generated_samples))
        all_samples_labels = torch.cat(
            (real_samples_labels, generated_samples_labels)
        )
        #
        # -----> Discriminator Training
        #
        # Zero the gradients of the discriminator (see note).
        discriminator.zero_grad()
        # Pass the mixed dataset through the discriminator.
        output_discriminator = discriminator(all_samples)
        # Calculate the loss for the discriminator. You calculate the loss␣
↪function using the output
        # from the model in output_discriminator and the labels in␣
↪all_samples_labels.
        loss_discriminator = loss_function(output_discriminator,␣
↪all_samples_labels)
        # You calculate the gradients to update the weights with.
        loss_discriminator.backward()
        # Update the discriminator's weights.
        optimizer_discriminator.step()
        #
        # -----> Generator Training
        #
        # Prepare new latent space samples for training the generator. You␣
↪store random data in
        # latent_space_samples, with a number of lines equal to batch_size. You␣
↪use two columns
        # since you're providing two-dimensional data as input to the generator.
        latent_space_samples = torch.randn((batch_size, 2))
        # Zero the gradients of the generator (see note).
        generator.zero_grad()
        # Generate fake samples using the generator.
        generated_samples = generator(latent_space_samples)
        # Pass the generated samples through the discriminator.
        output_discriminator_generated = discriminator(generated_samples)
        # Calculate the generator's loss. You calculate the loss function using␣
↪the output of the classification
        # system stored in output_discriminator_generated and the labels in␣
↪real_samples_labels,
        # which are all equal to 1.
        loss_generator = loss_function(output_discriminator_generated,␣
↪real_samples_labels)
```

```
        # Calculate the gradients to update the weights with.
        loss_generator.backward()
        # Update the generator's weights.
        optimizer_generator.step()

        # Optionally print the losses every 10 epochs.
        if epoch % 10 == 0 and n == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")
            fig = plt.figure()
            generated_samples_for_test = torch.zeros((batch_size, 2))
            generated_samples_for_test =␣
↪generator(latent_space_samples_for_test)
            generated_samples_for_test = generated_samples_for_test.detach()
            plt.plot(generated_samples_for_test[:, 0],␣
↪generated_samples_for_test[:, 1], ".")
            plt.savefig('./pics/gan_' + str(epoch).zfill(3))
            plt.close(fig)
```

```
Epoch: 0 Loss D.: 0.22915007174015045
Epoch: 0 Loss G.: 2.0208818912506104
Epoch: 10 Loss D.: 0.6539108753204346
Epoch: 10 Loss G.: 1.1530311107635498
Epoch: 20 Loss D.: 0.5917414426803589
Epoch: 20 Loss G.: 0.7660138607025146
Epoch: 30 Loss D.: 0.5403224229812622
Epoch: 30 Loss G.: 0.7709020376205444
Epoch: 40 Loss D.: 0.6078189611434937
Epoch: 40 Loss G.: 0.785312294960022
Epoch: 50 Loss D.: 0.6755154132843018
Epoch: 50 Loss G.: 0.7234432697296143
Epoch: 60 Loss D.: 0.6871098279953003
Epoch: 60 Loss G.: 0.7104587554931641
Epoch: 70 Loss D.: 0.6785175800323486
Epoch: 70 Loss G.: 0.7387452125549316
Epoch: 80 Loss D.: 0.6626092195510864
Epoch: 80 Loss G.: 0.7252268195152283
Epoch: 90 Loss D.: 0.628010094165802
Epoch: 90 Loss G.: 0.912176787853241
Epoch: 100 Loss D.: 0.756670355796814
Epoch: 100 Loss G.: 0.7376134395599365
Epoch: 110 Loss D.: 0.6580151915550232
Epoch: 110 Loss G.: 0.8232095241546631
Epoch: 120 Loss D.: 0.6767460107803345
Epoch: 120 Loss G.: 0.8058826923370361
Epoch: 130 Loss D.: 0.695110023021698
Epoch: 130 Loss G.: 0.6714414954185486
```

```
Epoch: 140 Loss D.: 0.6680338978767395
Epoch: 140 Loss G.: 0.7798112034797668
Epoch: 150 Loss D.: 0.6465719938278198
Epoch: 150 Loss G.: 1.059089183807373
Epoch: 160 Loss D.: 0.7089145183563232
Epoch: 160 Loss G.: 0.7115235328674316
Epoch: 170 Loss D.: 0.7168824076652527
Epoch: 170 Loss G.: 0.6684724688529968
Epoch: 180 Loss D.: 0.6568028926849365
Epoch: 180 Loss G.: 0.7500876188278198
Epoch: 190 Loss D.: 0.7503793835639954
Epoch: 190 Loss G.: 0.6951805353164673
Epoch: 200 Loss D.: 0.6927289962768555
Epoch: 200 Loss G.: 0.684615433216095
Epoch: 210 Loss D.: 0.625089168548584
Epoch: 210 Loss G.: 0.8152226209640503
Epoch: 220 Loss D.: 0.6968247294425964
Epoch: 220 Loss G.: 0.7450491786003113
Epoch: 230 Loss D.: 0.6922600269317627
Epoch: 230 Loss G.: 0.7031933665275574
Epoch: 240 Loss D.: 0.6187373995780945
Epoch: 240 Loss G.: 0.8768698573112488
Epoch: 250 Loss D.: 0.6453538537025452
Epoch: 250 Loss G.: 0.8081855773925781
Epoch: 260 Loss D.: 0.6491449475288391
Epoch: 260 Loss G.: 0.7140579223632812
Epoch: 270 Loss D.: 0.6024165749549866
Epoch: 270 Loss G.: 1.1036250591278076
Epoch: 280 Loss D.: 0.7330852746963501
Epoch: 280 Loss G.: 0.621140718460083
Epoch: 290 Loss D.: 0.6804800033569336
Epoch: 290 Loss G.: 0.7426090240478516
Epoch: 300 Loss D.: 0.6596154570579529
Epoch: 300 Loss G.: 1.0016937255859375
Epoch: 310 Loss D.: 0.6974169015884399
Epoch: 310 Loss G.: 0.6844212412834167
Epoch: 320 Loss D.: 0.642008900642395
Epoch: 320 Loss G.: 1.022351622581482
Epoch: 330 Loss D.: 0.6682328581809998
Epoch: 330 Loss G.: 0.726664125919342
Epoch: 340 Loss D.: 0.6491564512252808
Epoch: 340 Loss G.: 0.811085045337677
Epoch: 350 Loss D.: 0.6824460029602051
Epoch: 350 Loss G.: 0.7667980790138245
Epoch: 360 Loss D.: 0.7094178199768066
Epoch: 360 Loss G.: 0.8411445617675781
Epoch: 370 Loss D.: 0.6902003884315491
Epoch: 370 Loss G.: 0.7468070387840271
```

```
Epoch: 380 Loss D.: 0.6598289012908936
Epoch: 380 Loss G.: 0.7377553582191467
Epoch: 390 Loss D.: 0.6602611541748047
Epoch: 390 Loss G.: 0.8720359802246094
Epoch: 400 Loss D.: 0.5645396113395691
Epoch: 400 Loss G.: 1.1932969093322754
Epoch: 410 Loss D.: 0.6717198491096497
Epoch: 410 Loss G.: 0.778720498085022
Epoch: 420 Loss D.: 0.6909223794937134
Epoch: 420 Loss G.: 0.7514491081237793
Epoch: 430 Loss D.: 0.6869509220123291
Epoch: 430 Loss G.: 0.7285788655281067
Epoch: 440 Loss D.: 0.6888507008552551
Epoch: 440 Loss G.: 0.6843973398208618
Epoch: 450 Loss D.: 0.6472446322441101
Epoch: 450 Loss G.: 0.9175939559936523
Epoch: 460 Loss D.: 0.6482131481170654
Epoch: 460 Loss G.: 0.7945737242698669
Epoch: 470 Loss D.: 0.7267916798591614
Epoch: 470 Loss G.: 0.680546760559082
Epoch: 480 Loss D.: 0.7020904421806335
Epoch: 480 Loss G.: 0.6848165392875671
Epoch: 490 Loss D.: 0.6338849663734436
Epoch: 490 Loss G.: 0.7811532616615295
Epoch: 500 Loss D.: 0.7438719868659973
Epoch: 500 Loss G.: 0.6866573095321655
Epoch: 510 Loss D.: 0.6136643290519714
Epoch: 510 Loss G.: 0.9653654098510742
Epoch: 520 Loss D.: 0.6929857730865479
Epoch: 520 Loss G.: 0.6530900597572327
Epoch: 530 Loss D.: 0.6581246256828308
Epoch: 530 Loss G.: 0.8307712078094482
Epoch: 540 Loss D.: 0.6496720910072327
Epoch: 540 Loss G.: 0.7548527121543884
Epoch: 550 Loss D.: 0.7207420468330383
Epoch: 550 Loss G.: 0.7162513136863708
Epoch: 560 Loss D.: 0.6421326398849487
Epoch: 560 Loss G.: 0.764049232006073
Epoch: 570 Loss D.: 0.7293415069580078
Epoch: 570 Loss G.: 0.6839751601219177
Epoch: 580 Loss D.: 0.6861189603805542
Epoch: 580 Loss G.: 0.7147951126098633
Epoch: 590 Loss D.: 0.6365147233009338
Epoch: 590 Loss G.: 0.7669004797935486
Epoch: 600 Loss D.: 0.656179666519165
Epoch: 600 Loss G.: 0.8739244937896729
Epoch: 610 Loss D.: 0.6590447425842285
Epoch: 610 Loss G.: 0.7841903567314148
```

```
Epoch: 620 Loss D.: 0.7100046277046204
Epoch: 620 Loss G.: 0.7177774310112
Epoch: 630 Loss D.: 0.7048176527023315
Epoch: 630 Loss G.: 0.7329686284065247
Epoch: 640 Loss D.: 0.6711293458938599
Epoch: 640 Loss G.: 0.723970353603363
Epoch: 650 Loss D.: 0.6603408455848694
Epoch: 650 Loss G.: 0.7830702662467957
Epoch: 660 Loss D.: 0.5817649364471436
Epoch: 660 Loss G.: 1.0487637519836426
Epoch: 670 Loss D.: 0.7073400020599365
Epoch: 670 Loss G.: 0.7406259179115295
Epoch: 680 Loss D.: 0.6754567623138428
Epoch: 680 Loss G.: 0.8484858870506287
Epoch: 690 Loss D.: 0.6496427655220032
Epoch: 690 Loss G.: 0.7408666014671326
Epoch: 700 Loss D.: 0.6472513675689697
Epoch: 700 Loss G.: 0.7175557613372803
Epoch: 710 Loss D.: 0.6793556809425354
Epoch: 710 Loss G.: 0.6975767612457275
Epoch: 720 Loss D.: 0.5615934729576111
Epoch: 720 Loss G.: 1.2872427701950073
Epoch: 730 Loss D.: 0.697908878326416
Epoch: 730 Loss G.: 0.7828854918479919
Epoch: 740 Loss D.: 0.7237231731414795
Epoch: 740 Loss G.: 0.7492168545722961
Epoch: 750 Loss D.: 0.6575385928153992
Epoch: 750 Loss G.: 0.7059210538864136
Epoch: 760 Loss D.: 0.6291394233703613
Epoch: 760 Loss G.: 0.7493021488189697
Epoch: 770 Loss D.: 0.6642335653305054
Epoch: 770 Loss G.: 0.7083559632301331
Epoch: 780 Loss D.: 0.6127330660820007
Epoch: 780 Loss G.: 0.8346046209335327
Epoch: 790 Loss D.: 0.5470131635665894
Epoch: 790 Loss G.: 1.1161526441574097
Epoch: 800 Loss D.: 0.6756965517997742
Epoch: 800 Loss G.: 0.7905111312866211
Epoch: 810 Loss D.: 0.6669834852218628
Epoch: 810 Loss G.: 0.9834491014480591
Epoch: 820 Loss D.: 0.6420913338661194
Epoch: 820 Loss G.: 0.7505192160606384
Epoch: 830 Loss D.: 0.6602954864501953
Epoch: 830 Loss G.: 0.8223362565040588
Epoch: 840 Loss D.: 0.6735382080078125
Epoch: 840 Loss G.: 0.7241675853729248
Epoch: 850 Loss D.: 0.6928781270980835
Epoch: 850 Loss G.: 0.754580557346344
```

```
Epoch: 860 Loss D.: 0.631832480430603
Epoch: 860 Loss G.: 0.8492210507392883
Epoch: 870 Loss D.: 0.7169116139411926
Epoch: 870 Loss G.: 0.6906177401542664
Epoch: 880 Loss D.: 0.66900235414505
Epoch: 880 Loss G.: 0.7549993991851807
Epoch: 890 Loss D.: 0.6515700817108154
Epoch: 890 Loss G.: 0.9340789318084717
Epoch: 900 Loss D.: 0.6684414744377136
Epoch: 900 Loss G.: 0.8839318156242371
Epoch: 910 Loss D.: 0.6623920202255249
Epoch: 910 Loss G.: 0.6620436310768127
Epoch: 920 Loss D.: 0.6723664999008179
Epoch: 920 Loss G.: 0.7301745414733887
Epoch: 930 Loss D.: 0.6887557506561279
Epoch: 930 Loss G.: 0.7011988759040833
Epoch: 940 Loss D.: 0.7106810808181763
Epoch: 940 Loss G.: 0.6798657774925232
Epoch: 950 Loss D.: 0.6335510015487671
Epoch: 950 Loss G.: 0.9814509153366089
Epoch: 960 Loss D.: 0.6941967010498047
Epoch: 960 Loss G.: 0.7276195883750916
Epoch: 970 Loss D.: 0.6053673624992371
Epoch: 970 Loss G.: 0.7381106615066528
Epoch: 980 Loss D.: 0.6966307163238525
Epoch: 980 Loss G.: 0.6933261156082153
Epoch: 990 Loss D.: 0.6891751289367676
Epoch: 990 Loss G.: 0.6875872611999512
```

Checking the Samples Generated by the GAN

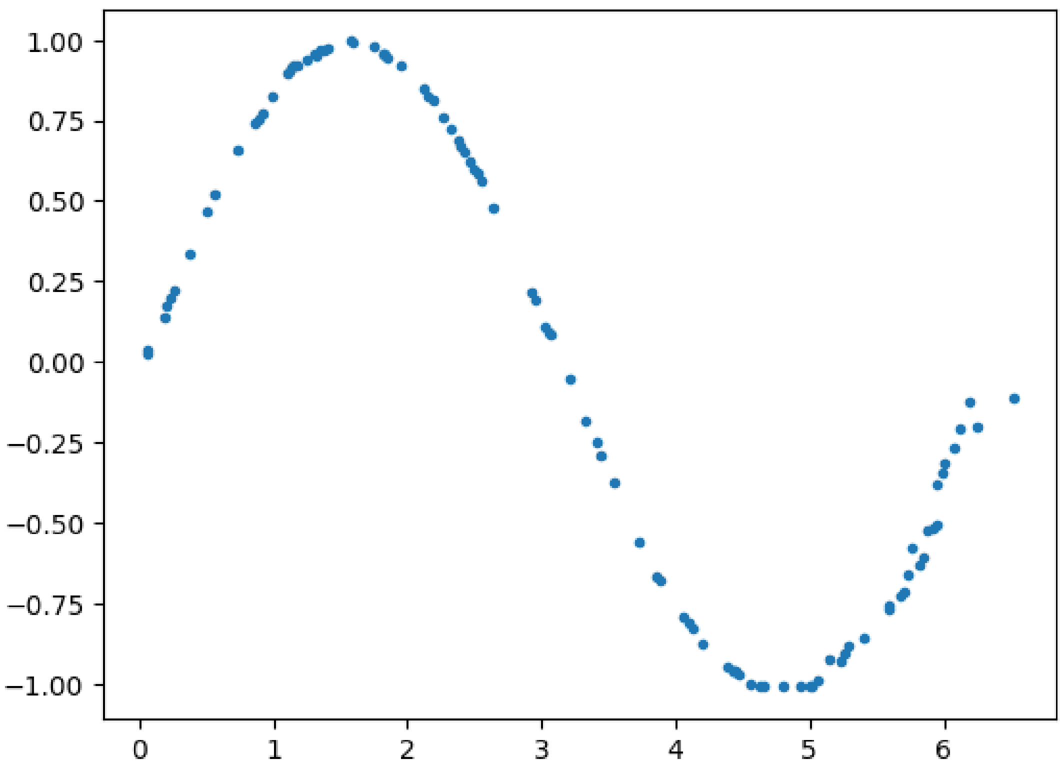
Generative adversarial networks are designed to generate data. So, after the training process is finished, you can get some random samples from the latent space and feed them to the generator to obtain some generated samples:

[25]:
```
latent_space_samples = torch.randn(100, 2)
generated_samples    = generator(latent_space_samples)
```

Then you can plot the generated samples and check if they resemble the training data. Before plotting the generated_samples data, you'll need to use `.detach()` to return a tensor from the PyTorch computational graph. The `detach()` method in PyTorch is like telling PyTorch to not worry about remembering the steps you took to create it. When training models, PyTorch keeps track of how you arrived at a result so it can learn from it. Sometimes, though, you just want to look at the result without changing the learning. Using `detach()` is like saying, "This part is just for showing, no need to learn from it." It helps when you're checking your work without affecting the learning process.

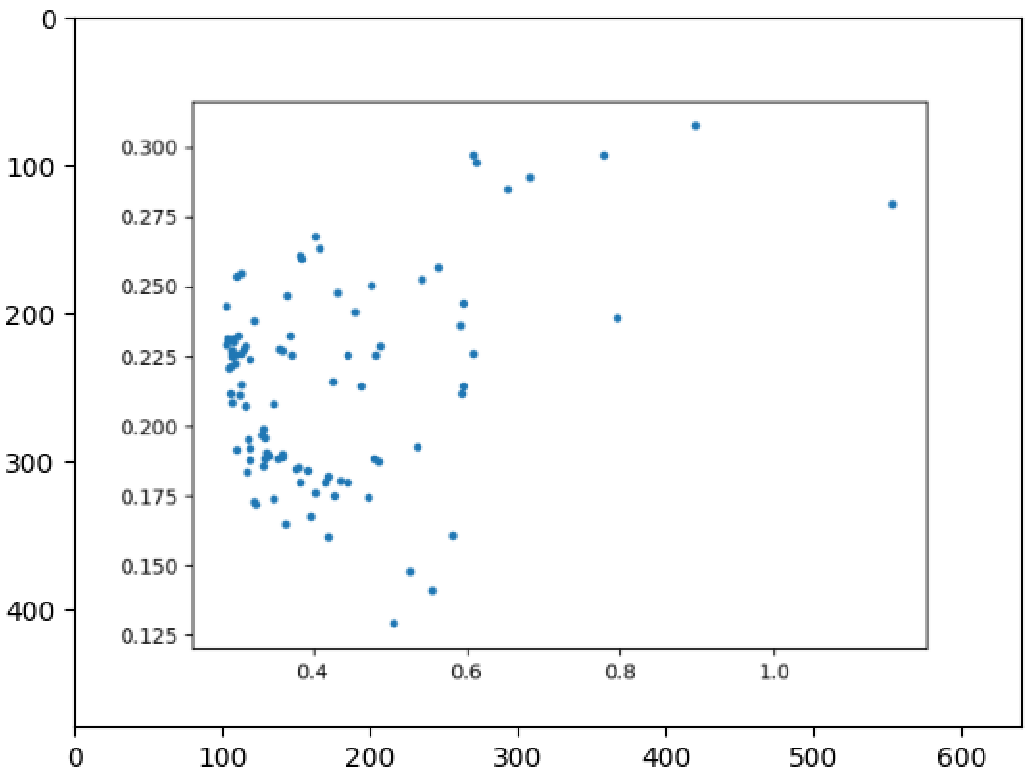
[26]:
```
generated_samples = generated_samples.detach()
plt.plot(generated_samples[:, 0], generated_samples[:, 1], ".")
```

[26]: [<matplotlib.lines.Line2D at 0x1dc781d0be0>]



[11]:
```
import matplotlib.animation as animation
from IPython.display import HTML
import os

# Assuming you have images stored in 'path/to/images' directory
image_files = [os.path.join('./pics', img) for img in sorted(os.listdir('./
 ↪pics')) if img[:4]=='gan_']

fig, ax = plt.subplots()

def animate(i):
    img = plt.imread(image_files[i])
    ax.clear()  # Clear the previous image
    ax.imshow(img)

ani = animation.FuncAnimation(fig, animate, frames=len(image_files),␣
 ↪interval=200)

# Display the animation in Jupyter
HTML(ani.to_jshtml())
```

```
[11]: <IPython.core.display.HTML object>
```



1.2 Reference and Credits

[1] - Renato Candido, "Generative Adversarial Networks: Build Your First Models", RealPython

1.3 Appendix

```
[ ]: x = torch.rand(4, requires_grad=True)
     w = torch.rand(4, requires_grad=True)

     y = x @ w  # inner-product of x and w
     z = y ** 2  # square the inner product
```

```
[ ]: z.backward()  # ask pytorch to trace back the computation of z
```

```
[ ]: print(w.grad) # the resulting gradient of z w.r.t w
```

```
[ ]: # since dz/dy = 2*y and dy/dw = x ...
     print(2*y*x)
```

- ref1
- ref2